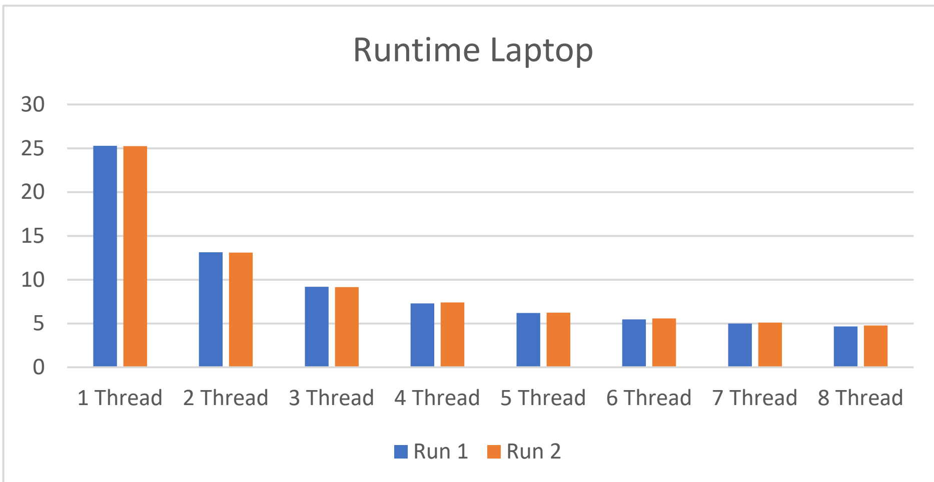
Project 4 CpE142

Jeremy Shaw

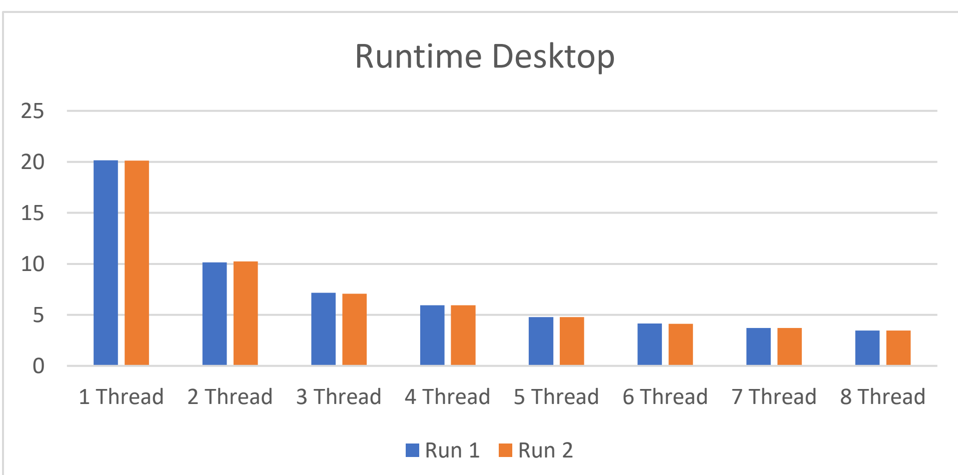
Professor Faroughi

Data (in seconds), first run is laptop, second run was done later on a desktop.

1 Thread	2 Thread	3 Thread	4 Thread	5 Thread	6 Thread	7 Thread	8 Thread
25.28	13.13	9.18	7.3	6.19	5.46	4.99	4.65
25.24	13.08	9.16	7.39	6.25	5.56	5.11	4.79



1 Thread	2 Thread	3 Thread	4 Thread	5 Thread	6 Thread	7 Thread	8 Thread
20.15	10.16	7.17	5.94	4.78	4.14	3.72	3.45
20.13	10.23	7.08	5.95	4.77	4.13	3.72	3.47



Notes:

On the laptop, Windows 10 Task Manager noted 100% CPU utilization at > 3 threads.
Laptop: Thinkpad X1 Carbon. Intel i5 8350u 16GB LPDDR3 2133Mhz effective rate.

On the desktop, Windows 10 Task Manager noted 100% CPU utilization at > 6 threads.
Desktop: DIY PC. Intel i7 7700k 32GB DDR4 2400Mhz effective rate.

Speedup results from threading:

Reduced by (ratio) => result is derived by comparing the current n Thread average to the n-1 Thread average.

Faster by (ratio) => (a.k.a. speedup) result is derived by comparing the current n Thread average to the n-1 Thread average.

Laptop

	1 Thread	2 Thread	3 Thread	4 Thread	5 Thread	6 Thread	7 Thread	8 Thread
Run 1	25.28	13.13	9.18	7.3	6.19	5.46	4.99	4.65
Run 2	25.24	13.08	9.16	7.39	6.25	5.56	5.11	4.79
Average	25.26	13.105	9.17	7.345	6.22	5.51	5.05	4.72
Reduced by (ratio)		0.5188	0.69973	0.80098	0.84683	0.88585	0.91652	0.93465
Faster by (ratio)		1.92751	1.42912	1.24847	1.18087	1.12886	1.09109	1.06992

Desktop

	1 Thread	2 Thread	3 Thread	4 Thread	5 Thread	6 Thread	7 Thread	8 Thread
Run 1	20.15	10.16	7.17	5.94	4.78	4.14	3.72	3.45
Run 2	20.13	10.23	7.08	5.95	4.77	4.13	3.72	3.47
Average	20.14	10.195	7.125	5.945	4.775	4.135	3.72	3.46
Reduced by (ratio)		0.50621	0.69887	0.83439	0.8032	0.86597	0.89964	0.93011
Faster by (ratio)		1.97548	1.43088	1.19849	1.24503	1.15478	1.11156	1.07514

Here, we can see the Desktop gains a ~1.98x speedup when going from 1 thread to 2 threads. We can also see the Desktop gains a total ~1.71x speedup when going from 2 threads to 4 threads.

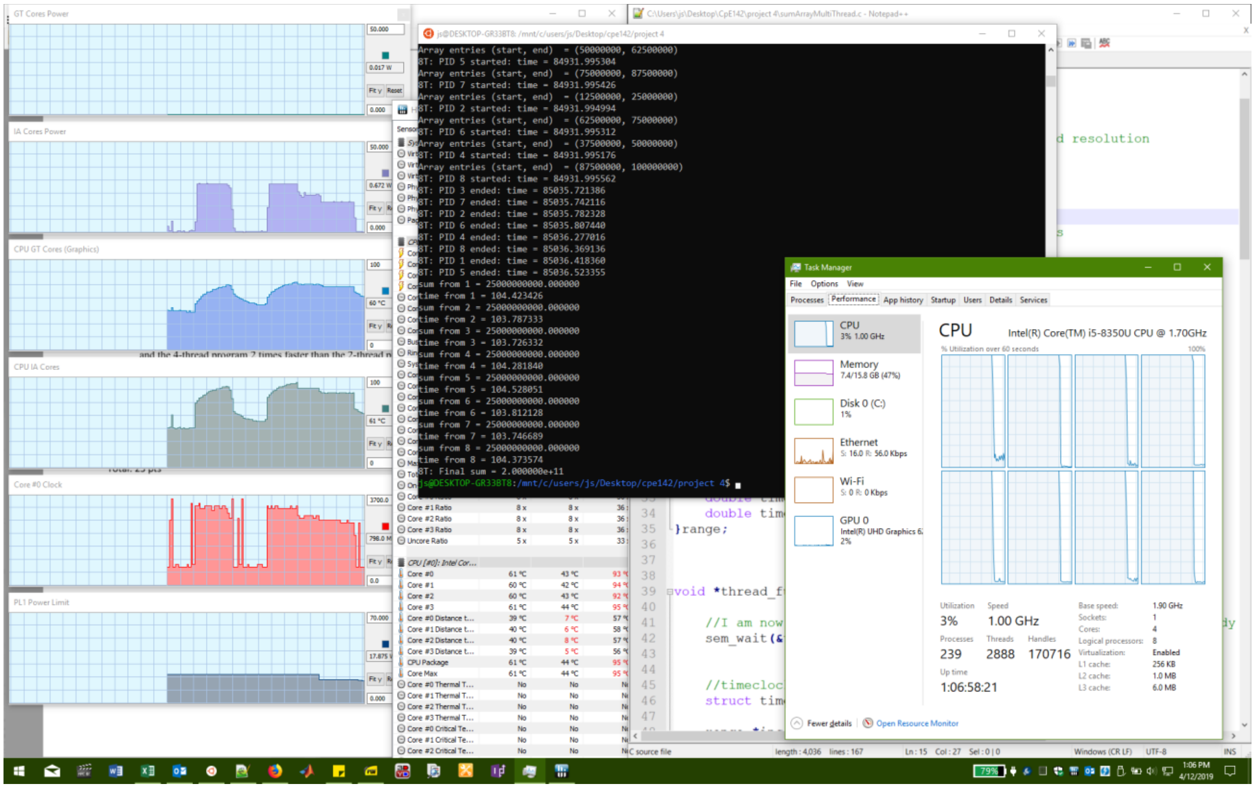
This is a lessor speedup (a decline in speedups can be observed) than 1T -> 2T. Part of this may be due to bottlenecks within the system (particularly w.r.t. the memory subsystem, which was not benchmarked in this Project). Additionally, when going above 2 threads, the CPU scheduler still has to take care of existing system processes. Those existing processes have a greater chance of interrupting the Project's threads when the scheduler is no longer capable of finding an "empty core" to offload work.

As a result, I believe a combination of system bottlenecks and thread collisions with other processes are the primary factors to declining performance gains.

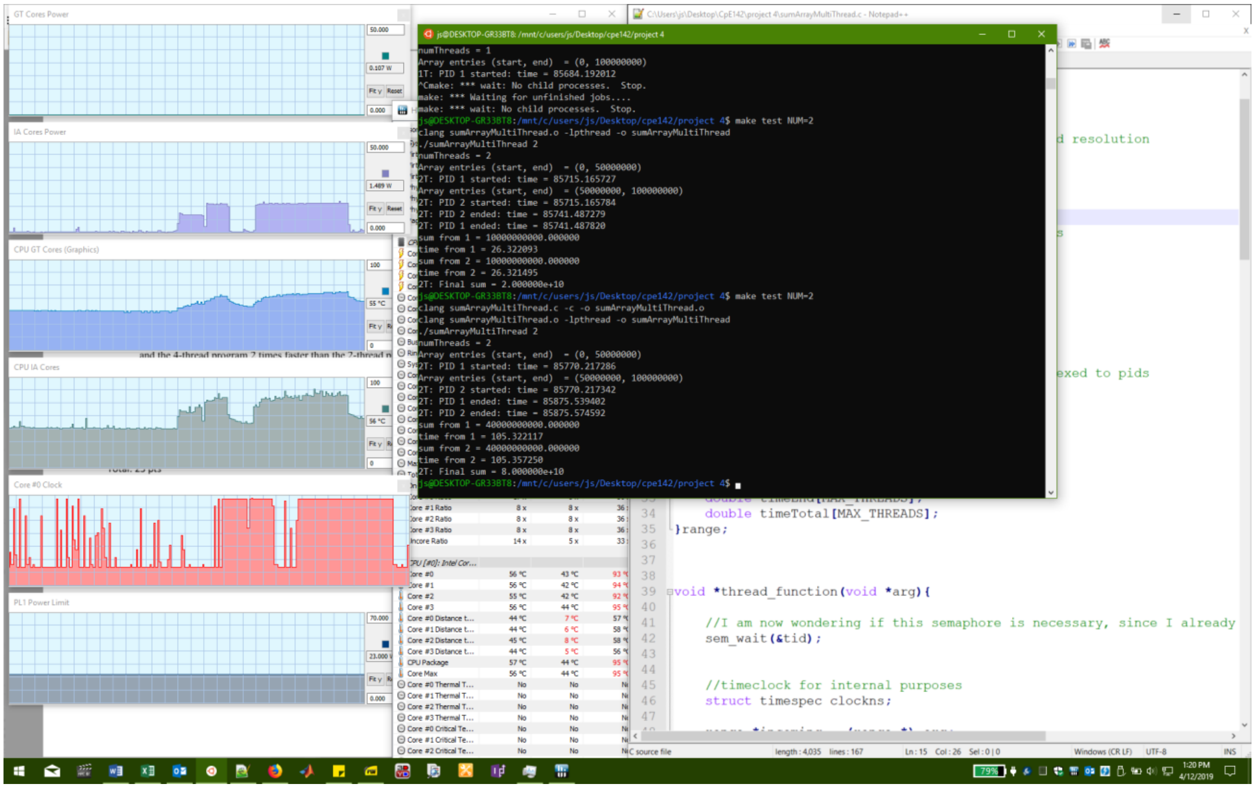
Appendix A: Long term performance

When using a laptop for sustained performance, there are some additional factors at play.

In a multithreaded, high load application, the CPU in my laptop (i5 8350U – Intel 4 Core, 8 Thread) is only capable of maintaining a short term (<30 second) ~3.2Ghz all core clock. After which, if the load is still present, it further restricts itself to ~2.8Ghz across all cores. After ~47 seconds, the laptop further drops down to ~2.55Ghz across all cores.



However, with a very light threading (using a smaller iteration count; also 2 threads instead of 8):

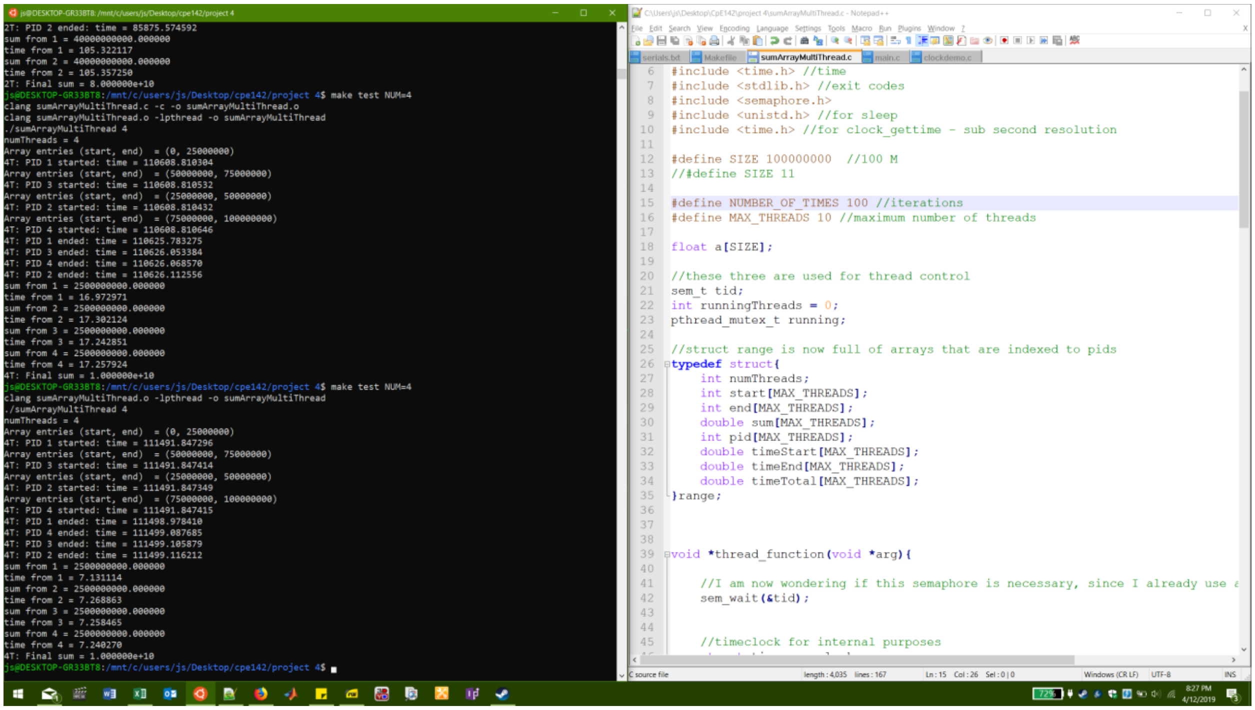


The laptop (CPU + heatsink + fan) is now capable of maintaining 3.6Ghz for the entire 105 second run. This is Intel's "Turbo Boost" technology at work, alongside regular thermal protection measures. It does make comparisons of lightly threaded and heavily threaded work a bit more difficult on this laptop. Maximum Turbo Boost clock is 3.6Ghz for this SKU.

It appears the DIY Desktop (DIY assembly with off-the-shelf parts) is not encumbered in this way. It appears capable of maintaining it's 4 core turbo boost (4.5Ghz cores) indefinitely. This was the intent behinds the parts selection for this desktop (parts not listed).

Appendix B: Laptop battery power

The laptop I use has a small, 3 cell battery. The laptop CPU alone can draw 27W under full load, which is a large strain on the battery, and other devices like the screen also contribute ~4-13W (estimated from battery drain rate) depending on the brightness. As such, the laptop is incapable of maintaining full performance when on battery power.



Here, two loops are run with identical settings. However, the first run is on battery power, and the second run is on mains power.

Appendix C: Laptop performance considerations:

In light of Appendices A and B, the array size and iterations have been chosen to be 100,000,000 and 100, respectively. When the laptop is plugged in, these chosen values allow the entire program to complete within the ~< 30second “full power” allowance of the laptop.

Appendix D: Source code of “user controllable multithreading program”

sumArrayMultiThread.c

//Jeremy Shaw CpE142
//Project 4
//Professor Faroughi
#include <pthread.h> //pthread
#include <stdio.h> //io

```
#include <time.h> //time
#include <stdlib.h> //exit codes
#include <semaphore.h>
#include <unistd.h> //for sleep
#include <time.h> //for clock_gettime - sub second resolution

#define SIZE 100000000  //100 M
//#define SIZE 11

#define NUMBER_OF_TIMES 100 //iterations
#define MAX_THREADS 10 //maximum number of threads

float a[SIZE];

//these three are used for thread control
sem_t tid;
int runningThreads = 0;
pthread_mutex_t running;

//struct range is now full of arrays that are indexed to pids
typedef struct{
	int numThreads;
	int start[MAX_THREADS];
	int end[MAX_THREADS];
	double sum[MAX_THREADS];
	int pid[MAX_THREADS];
	double timeStart[MAX_THREADS];
	double timeEnd[MAX_THREADS];
	double timeTotal[MAX_THREADS];
}range;



void *thread_function(void *arg){

	//I am now wondering if this semaphore is necessary, since I already use another scheme
	sem_wait(&tid);


	//timeclock for internal purposes
	struct timespec clockns;

	range *incoming = (range *) arg;
	int numThreads;
	double sum, timeStart, timeEnd;
	int start, end, pid, index, quantum;

	int ttid = 0;
```

```
sem_getvalue(&tid, &ttid);
//printf("tid = %d\n", ttid);

numThreads = incoming->numThreads;
pid = numThreads-ttid;
index = pid - 1;

incoming -> pid[index] = pid;

quantum = SIZE / numThreads;

start = quantum * (pid - 1);
end = quantum * pid;

if(end == SIZE - 1){
	end = SIZE;
}

//starting time
clock_gettime(CLOCK_MONOTONIC, &clockns); //using CPU unset time
timeStart = (double)clockns.tv_sec + (double) clockns.tv_nsec / 1000000000;
incoming -> timeStart[index] = timeStart;

//printf("Array entries (start, end)  = (%d, %d)\n", start, end);
printf("%dT: PID %d started: time = %f\n", numThreads, pid, timeStart);

sum = 0;
for(int j = 0; j < NUMBER_OF_TIMES; j++){
	for (int i = start; i < end; i++){
		sum += a[i];
		//printf("a[%d] = %f\n", i, a[i]);
	}
}

incoming -> sum[index] = sum;   //save the result from the PE

clock_gettime(CLOCK_MONOTONIC, &clockns);
timeEnd = (double)clockns.tv_sec + (double) clockns.tv_nsec / 1000000000;
incoming -> timeEnd[index] = timeEnd;

printf("%dT: PID %d ended: time = %f\n", numThreads, pid, timeEnd);

incoming -> timeTotal[index] = timeEnd - timeStart;

sem_post(&tid);

pthread_mutex_lock(&running);
runningThreads--;
```

```
	pthread_mutex_unlock(&running);

	return NULL;
}

int main (int argc, char *argv[])
{
	pthread_t threadID;
	void *exit_status;
	range worker;
	double sum = 0;
	int numThreads = 0;

	pthread_mutex_init(&running, NULL);

	if(argc != 2){
		fprintf(stderr, "Wrong number of arguements\n");
		exit(EXIT_FAILURE);
	}
	else{
		numThreads = atoi(argv[1]);
		printf("numThreads = %d\n", numThreads);
		if(numThreads > MAX_THREADS){
			fprintf(stderr, "Too many threads, max %d\n", MAX_THREADS);
			exit(EXIT_FAILURE);
		}
	}

	for(int i = 0; i < SIZE; i++){ //initialize array
		a[i] = 1;
	}

	//allows for a max total of threads
	sem_init(&tid, 0, numThreads);

	//thread_function(&worker2);

	worker.numThreads = numThreads;
	for(int i = 0; i < numThreads; i++){
		pthread_mutex_lock(&running);
		pthread_create(&threadID, NULL, thread_function, &worker);
		runningThreads++;
		pthread_mutex_unlock(&running);
	}

	//wait until threads are done, then continue.
	while(runningThreads){
		sleep(1);
```

```
	}

	//this is not going to consistently work for more than 1 spawned thread.
	//pthread_join(threadID, &exit_status); //wait for the 1st thread to end

	for(int i = 0; i < numThreads; i++){
		sum += worker.sum[i];
		printf("sum from %d = %f\n", i+1, worker.sum[i]);
		printf("time from %d = %f\n", i+1, worker.timeTotal[i]);
	}

	double lowTime = 0;
	double highTime = 0;

	lowTime = worker.timeStart[0];
	highTime = worker.timeEnd[0];

	for(int i = 0; i < numThreads; i++){
			if(lowTime > worker.timeStart[i]){
				lowTime = worker.timeStart[i];
			}
			if(highTime < worker.timeEnd[i]){
				highTime = worker.timeEnd[i];
			}
	}

	//printf("2T: PID 0 ended: time = %f\n", (double) time(NULL));

	printf("%dT: Final sum = %e\n", numThreads, sum);
	printf("%dT: Total time = %f\n", numThreads, highTime - lowTime);

	return 0;
}
```

Makefile

```
CC = clang

NUM?=4

all: sumArrayMultiThread.o
	clang sumArrayMultiThread.o -lpthread -o sumArrayMultiThread

sumArrayMultiThread.o: sumArrayMultiThread.c
	clang sumArrayMultiThread.c -c -o sumArrayMultiThread.o
```

```
test: all
	./sumArrayMultiThread $(NUM)

clean:
	rm *.o sumArrayMultiThread
```

The next program was not used by the complete project. This is a small program that demonstrates a nanosecond-accurate clock similar to the one ultimately used in sumArrayMultiThread.c.

clockdemo.c

```
//Jeremy Shaw

#include <time.h>
#include <stdio.h>

#define ITERATIONS 1000

int main(){
	struct timespec clockns;

	/*
	CLOCK_PROCESS_CPUTIME_ID is proportionally related to realtime.
	There is a constant scalar value.
	CLOCK_MONOTONIC will be used for the actual Project 4 program.
	*/
	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &clockns);
	printf("tv_sec = %ld tv_nsec = %ld\n", clockns.tv_sec, clockns.tv_nsec);
	int array[1000000];
	long sum = 0;

	for(int i = 0; i < 1000000; i++){
		array[i] = 1;
	}

	for(int i = 0; i < ITERATIONS; i++){
		for(int j = 0; j < 1000000; j++){
			sum += array[j];
		}

	}

	printf("sum = %lu\n", sum);
	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &clockns);
	printf("tv_sec = %ld tv_nsec = %ld\n", clockns.tv_sec, clockns.tv_nsec);
```

```
	long seconds = clockns.tv_sec;
	long nanoseconds = clockns.tv_nsec;

	double totalTime = (double)seconds + (double)nanoseconds/1000000000;
	printf("Total Time = %f\n", totalTime);
	return 0;
}
```